برمجة هدفية التوجه بلغة ++C

ترجمة وإعداد:أ.فهد أحمد آل قاسم
fhdalqasem@yahoo.com

الجزء الأكبر مأخوذ من كتاب :
(Introduction To Object Oriented Programming In C++)
لمؤلفه Yashavant Kanetkar من منشورات BPB PUBLICATION

-١-

الفئات والكائنات (Class & Object):

مقدمة:

يعتبر مفهوم الفئة class واحدا من أفضل ميزات لغة سي++ (C++) التي لم تكن موجودة في لغة (C)، الفئة هي مجموعة من البيانات Data والدوال (Functions) التي تعمل على هذه البيانات، أما الكائن (object) فهو تطبيق محجوز في الذاكرة يستخدم وفقا لتعريف الفئة النوع .

في لغة سي ++ (C++)لا يوجد فرق عملي بين التركيبات (structures) والفئات (classes)، خاصة بعد قابلية التركيبات لإحتواء دوال (Functions) ضمن متغيراتها كإضافة جديدة للغة سي ++ (C++)على لغة سي (C)، ولذلك فبإمكان كلا منهما الإستخدام تبادليا، لكن معظم مبرمجي لغة سي ++((C++)) يستخدمون التركيبات من أجل إحتواء البيانات فقط (كما كانت عليه في لغة سي C)، ويستخدمون الفئات للتعامل مع كلا من البيانات والدوال.

التصريح عن فئة (Declaration of Class)

التصريح عن الفئة يحدد أعضاءها من دوال وبيانات، كما يقوم بتحديد المدى(Member Scope) لكل عضو من أعضاء الفئة الشكل العام للتصريح عن الفئة كالتالي:

```
class class_name
{
        private:
                DataMembers declaration;
                        FunctionMembers declaration;

        public:
                DataMembers declaration;
                        FunctionMembers declaration;

};
```

إن الكلمة المحجوزة (class) تخبر المترجم(Compiler) أن ما يليها هو إسم فئة ومابعده هو تصريحات أعضاء تلك الفئة، وكما هو الحال مع التركيب فإن التصريح عن الاعضاء يحاط بحاصرتين وينتهي بفاصلة منقوطة.

أعضاء الفئة (class members): هي المكونات ذات الانواع المعروفة التي يتم التصريح عنها في جسم الفئة، وهي إما بيانات (Data) او دوال (Functions)، بعض المؤلفين يسمون الدالة الخاصة بالفئة او الكائن بالطريقة (method)، بينما سنقوم هنا بتسمية البيانات التابعة لفئة بأعضاء البيانات(Data Members) والتابعة لفئة بالأعضاء الدوال (Member functions).

أما الكلمتين المحجوزيتين (private)و(public) فهما وسيلة البرمجة الهدفية في تغليف الكائن والفئة (Encapsulation) أو ما يسمى بإخفاء البيانات (data hiding)، وهما مصطلحان يقصد بهما عملية أمنية البيانات وجعلها حصرية في النطاق المطلوب، فالأعضاء (من بيانات ودوال)التي تأتي بعد الكلمة (private) تكون أعضاء حصرية للإستخدام على مستوى الفئة وأعضائها من بيانات ودوال أيظا، أما الكلمة المحجوزة (public) فتعني ان

الأعضاء التالية غير حصرية الاستخدام، أي ان مدى الإستخدام والتعامل (scope) مدى عام، سواء على مستوى الدالة الرئيسية (Main) أو الفئات الاخرى، إن الوضع التلقائي هو الوضع الخاص المحلي (private) بالنسبة لمحتوى الفئة.

والآن لنرى بناء جملة تحتوي على فئة في المثال التالي:

```
class rectangle
        {       private:
                        int len,br;
                public:
                                void getdata();
                void setdata(int l,int b);
                void displaydata( );
                void area_peri( );
 };
```

لقد أنشأنا الآن نوع بيانات جديد إسمه (RECTANGLE)، يتكون نوع البيانات الجديد هذا من ستة أعضاء، عضوي البيانات (br)و(len) وهما من النوع العددي الصحيح، وأربعة أعضاء كلها إجراءات، كتبت التصريحات الخاصة بها ولم يتم تسجيل التعريف الخاص بعمل كل واحدة.

سنقوم فيما بعد بكتابة محتوى كل واحدة من الإجراءات/الدوال المصرح عنها في جسم الفئة(class body)، من الملاحظ ان الدوال جميعها معرفة في خانة الأنواع العامة(public)، بينما المتغيرات (البيانات) معرفة في جانب النوع ذي المدى الحصري على مستوى الفئة(private)، وهذه هي العادة الغالبة على مستخدمي البرمجة الهدفية، إذ ان المطلوب في الغالب هو كتابة أعضاء دوال تنفذ خارجيا وبيانات تستخدمها هذه الدوال .

وهذا لا يعني ان هناك قواعد تحتم كون البيانات الأعضاء في الفئة ذات مدى محلي، والدوال الأعضاء ذات مدى عام، إذ يمكن للمبرمج في أحيان أخرى التصريح عن أعضاء خاصة/محلية وبيانات عامة أو عن بيانات ودوال عامة حسب رغبة المبرمج.

إنشاء مثال/متغير عن الفئة (class instance):

إن نوع البيانات (float) مثلاً يعرف طريقة معينة للتعامل مع البيانات التي من ذلك النوع، ولو عرفنا متغيرا y من ذلك النوع لأستخدمنا الجملة :

```
Float y;
```

في هذه الجملة ندعو المتغير y بأنه مثال للنوع float يحجز موقعا في الذاكرة بذات مواصفات ذلك النوع (المسجلة مسبقا)، بنفس الطريقة نقوم بتعريف مثال (instance) للفئة التي نرغب بتعريف مثال عنها، مثلاً نقوم بتعريف مثال عن الفئة السابقة (فئة نوع البيانات) المعرف أعلاه بالإسم (rectangle)، وذلك بنفس الطريقة:

```
rectangle r1,r2;
```

إننا بهذه الجملة قد عرفنا (متغيرين r1 و r2) من النوع (rectangle)، وكل متغير هو مثال عن الفئة المعرفة أعلاه، أي أننا نعرف كائن يقوم بنفس الدور المرسوم له في تعريف الفئة (rectangle)، وهي طريقة مشابهة للتصريح عن متغير كما تعودنا في المتغيرات الاساسية للغة ++c، ولكن الفارق هنا ان نوع البيانات معرف عن طريق المستخدم نفسه.

عندما نعرف متغير من نوع صحيح مثلا، فإنه يحجز في الذاكرة حيزا يسع ٢بايت من البيانات (يعتمد حجم نوع البيانات المحجوز على المترجم ونظام التشغيل المستخدم، ولنتذكر الدالة (sizeof())، وهذا يعني بنفس الطريقة انه عند تعريف كائن object فإنه يقوم بحجز حيز من الذاكرة، في حالة الكائن r1 مثلا فإن الحيز من الذاكرة يساوي مجموع الانواع القياسية المعرفة في التصريح العام عن الفئة، وعند تعريف الكائن الآخر r2 فإنه يتم حجز مساحة مشابهة تماما للكائن السابق بنفس الطريقة.

من المهم التأكيد على ان التصريح عن فئة **class** لا يؤدي لحجز اي نطاق فعلي في الذاكرة، وأن ذلك يحدث فقط عند تعريف كائن(**instance object**) من تلك الفئة.

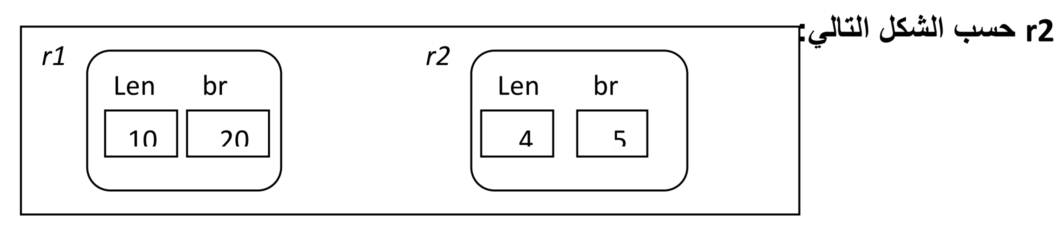
الوصول إلى أعضاء الفئة(Accessing class members):

إذا كان عضو الفئة من النطاق المحلي **private**، فإننا لا نستطيع الوصول إليه على مستوى الدالة الرئيسة **main()**، إن الاعضاء المحلية تكون قابلة الوصول لديها متاحة فقط على مستوى تعريف الفئة، فلإسناد قيم أو إطلاق قيم لأعضاء محلية (بيانات كانت او دوال) فإننا نستخدم الدوال الخاصة بالفئة نفسها.

ولإستخدام دالة عضو في كائن نستخدم إسم الكائن ملحوقا بنقطة **dot** ثم إسم ذلك الدالة **(object.function())** كما هو موضح في المثال التالي بخصوص الكائن **r1:**

```
r1.setdata(10,20);   r2.setdata(4,5);
```

إن المتغيرين **br** و **len** يحجزان موقعين في الذاكرة كمتغيرين صحيحين مرة ضمن الكائن **r1** ومرة ضمن الكائن **r2** حسب الشكل التالي:



تعريف (الأعضاء الدوال) للفئة:

بالإمكان تعريف وكتابة محتوى العضو الدالة في الفئة إما ضمن الحاصرتين في الفئة نفسها أو خارج حاصرتي الفئة حسب ما سوف توضحه الأمثلة التالية، لنعد لمثال المستطيل السابق، ولنعرف محتوى الدوال التي فيه ضمن حاصرتي الفئة ليكون شكل التصريح عن الفئة كالتالي :

```
Class rectangle
{
        private:
                int len,br;
        public:
                void getdata()
                {
                 cout<<endl<<"enter length and breadth";
                cin>>len>>br;
                }
                void setdata(int l,int b)
                {
                len=l;
                br=b;
                }
                void displaydata()
                {
                cout<<endl<<"length="<<len;
                cout<<endl<<"breadth="<<br;
                }
                void area_peri()
```

```
                {
                Int a,p;
                a=len*br;
                p=2*(len+br);
                cout<<endl<<"area="<<a;
                cout<<endl<<"perimeter="<<p;
                }
    };
```

ولكتابة الدوال خارج حاصرتي الفئة فإننا نستخدم المؤثر :: هذا المؤثر يعني أن الدالة على يساره هو عضو في الفئة المذكورة قبله على الصورة:

```
Return-type calss_name::function_name(argument lis)
{
        .....function body.....
}
```

حيث أن Return-type تعني نوع البيانات الذي يعيده الدالة (void أو int أو char أو غيره من ...)
calss_name إسم الفئة التي ينتمي إليها الدالة
:: المؤثر المذكور
function_name إسم الدالة المقصود كتابة محتواه.
argument lis قائمة المتغيرات المدخلة ضمن الدالة (إن وجدت).

تمرين :
أعد كتابة المثال السابق بطريقة تعريف الدوال خارج حاصرتي الفئة، مع أخذ الملاحظة رقم ١ أدناه بالإعتبار.
ملاحظات :

١. من أجل كتابة الدالة خارج حاصرتي التصريح عن الفئة class declaration، يجب كتابة تصريحات عن الدوال الأعضاء ضمن إطار التصريح عن الفئة كما هو موضح في أو مثال عن الفئات.

٢.الدوال أو الدوال المعرفة في الفئة ضمن الحاصرتين هي من النوع (inline)، أما الدوال المعرفة خارج إطار حاصرتي الفئة فليست من النوع inline، ولكي يتم تعريفها كدوال من تلك الصفة، يتم كتابة الكلمة المحجوزة inline قبل تعريف الدالة فيصبح الشكل العام أعلاه كالتالي:

```
Inline Return-type calss_name::function_name(argument lis) {}
```

ما هي الدوال من النوع inline ؟

٣. من المفيد جدا تعريف الدوال خارج إطار حاصرتي الفئة وذلك في حالة الفئات الكبيرة، إذ انه يتم عادة كتابة تصريحات الفئة ضمن ملف رأسي ((headfile(*.h) ويتم كتابة محتوى الدوال (التعريف) ضمن ملفات مصدرية (cpp.*) وذلك عند تأسيس المكتبات libraries المحتوية على عدد كبير من الفئات.

٤. الفئة المحلية (local class) هي الفئة التي يتم التصريح عنها داخل الدالة الرئيس (mian function)، ولا يصح في حالة التصريح عن فئة محلية أن يتم التصريح عن الدوال الاعضاء فقط من أجل التعريف خارج جسم الفئة.

الكائنات والدوال :

كما درسنا في الدوال functions، فإننا نعلم انها مجموعة أوامر يكتبها المستخدم، وتقبل مجموعة من المتغيرات، وتقوم بتنفيذ مجموعة الأوامر تلك، ثم تعيد(return) مجموعة من المتغيرات الناتجة.

نحن نعلم ان بعض الدوال لا تقبل متغيرات، كما أنه بالإمكان أن لا تعيد بعض الدوال اي متغير، إن تعامل الكائنات مع هذه الدوال يعتمد على مدى العضو (scope of the member) المستخدم.

أما بخصوص الدوال التي تقبل أو تعيد متغيرات، فهي تتعامل مع الكائنات .. ولكن كيف؟

تمرير كائن كمتغير في دالة:

كما هو الحال مع المتغيرات في الانواع القياسية المعروفة، فإن من الممكن تمرير كائن إلى دالة إما بالقيمة (by value) أو بالمرجع (by reference)، في المثال التالي يقوم البرنامج المكتوب بتعريف فئة ثم القيام بإستخدام دوال المكتبة (string.h) في دمج قيم كائنين من نوع تلك الفئة، (أي قيم عضوي بيانات في كائنين بالطبع):

```
#include <iostream.h>
#include <string.h>

Class str
{
private:
        char s[50];
public:
        void set (char *ss)
        {
        strcpy(s,ss);
        }
        void print()
        {
        cout<<s<<endl;
        }
        void concat (str s2)
        {
        strcat(s,s2.s);
        }
};
void main()
{
        str s1,s2;
        s1.set("hand in");
        s2.set("hand");
        s1.concat(s2);
        s1.print();
}
```

تحتوي الفئة str على متغير محلي هو عبارة عن سلسلة نصية (مصفوفة محارف array of characters)، وعلى ثلاثة دوال تقوم الدالة set() بقبول سلسلة نصية وخزنها (بنسخها) في المتغير المحلي، كما تقوم الدالة print() بطباعة محتوى السلسلة النصية، في حين تقوم الدالة concat() بدمج محتوى تلك السلسلة النصية لهذه الفئة(الكائن بالطبع) مع سلسلة نصية مشابهة لكائن من نفس نوع الفئة(اي انه يشترط ان يحتوي على سلسلة بنفس الاسم والنوع).

في الدالة الرئيس ()main يتم إسناد سلسلة نصية للكائن s1 عبر الدالة ()set ، ونفس العملية بالنسبة للكائن s2 ثم يتم في السطر الرابع إستخدام الدالة العضو في الكائن s1 لتنفيذ عملية دمج السلسلتين في الكائنين، ومن ثم في السطر الخامس طباعة محتوى السلسلة في الكائن s1 بعد الدمج.

نتيجة البرنامج السابق هي (hand in hand).

ولكن ما الذي سينتج؟ .. إذا تم إستبدال الكائن s2 بالكائن s1 في السطرين الرابع والخامس كالتالي:

```
s2.concat(s1);
s2.print();
```

ملاحظة : لا توجد علاقة بين الكائن s2 المعرف في أول سطر بالدالة الرئيس ()main، والكائن بنفس الاسم الممرر في تعريف الدالة void concat (str s2)، حتى لو تم إستخدام نفس المحارف للتسمية، إذ أن الكائن في تصريح تلك الدالة هو مجرد كائن وهمي يستخدم (كما هو الحال في الدوال) لتوضيح الإجراءات المستخدمة في حال تم تمرير كائن من النوع str، وذلك دون ان يتم إعتباره كائنا فعليا، بينما الكائن s2 المستخدم في الدالة الرئيس هو كائن حقيقي من النوع str.

تمرير مصفوفة كائنات كمتغيرات في دالة:

بالتأكيد كما في التركيب نستطيع إنشاء مصفوفة كائنات، مستخدمين نفس طريقة بناء الجملة في التصريح عن مصفوفة أعداد صحيحية أو حقيقية(integers or floats)، سوف يقوم البرنامج التالي بتعريف دالة عادية تقبل مصفوفة كائنات ممره إليه:

```
class sample
{
private:
        int i;
public:
        void set(int ii)
        {
                i=ii;
        }
        void print()
        {
                cout<<endl<<i<<endl;
        }
};
void show(sample *p)//non-memeber function..
{
        for (int j=0;j<5;j++)
        p[j].print();
}
  void main()
{
        sample s[5];
        int x;
```

```
        for(int j=0;j<5;j++)
        {
            cin>>x;
            s[j].set(x);
        }

        show(s);
    }
```

++++++++++++++++++++++

- ٢ -

الباني والهادم (Constructors & Destructors)

مقدمة:

قبل الدخول في تفاصيل الباني والهادم دعونا نتذكر الطرق المختلفة للتصريح عن متغير عادي، تابع لأحدالأنواع القياسية المعروفة :

```
int x;
```

مثلا المتغير x يحمل قيم مختلفة حسب المترجم compiler المستخدم، إذا يمكن ان تكون صفرا أو أقل قيم النوع الصحيح MIN_INT وهذه القيمة أيضا تعتمد على الحيز الذي تحتله الاعداد الصحيحة في الذاكرة حسب المترجم ونظام التشغيل المستخدم.

من العادات البرمجية الجيدة عملية إطلاق initialization قيم إبتدائية initial values للمتغير، وذلك بإحدى طريقتين إما بعد التصريح عنه :

```
int x;
x=5;
```

أو بنفس عبارة التصريح بأحد طريقتين إما:

```
int x=5;
```

أو بطريقة مكافئة :

```
int x (5);
```

إن فكرة الباني هي فكرة مشابهة لما سبق إلا ان خصوصية الفئات تتطلب مفاهيما أعقد من ذلك بكثير.

دعونا نتأمل هذا البرنامج البسيط:

```
#include <iostream.h>
class stack
{
private:
int i;
int a[10];
public:
void init()
{
i=0;
}
void push (int d)
{
a[i]=d;
i++;
}
void print()
{
for (int j=0;j<I;j++)
cout<<endl<<a[j];
}
};
void main()
{
```

```
stack s;
s.init();
s.push(10);
s.push(20);

s.print();
}
```

هذا البرنامج يعرف في المتغيرات العامة global، فئة تحت الإسم stack، تتكون هذه الفئة الـclass من احدى عشر عضو بيانات كلها صحيحة (عدد صحيح ومصفوفة من عشرة اعداد)، ويتكون كذلك من مجموعة من الأعضاء الدوال في هذه الفئة stack تعمل كالتالي:

- push : تعبئ أعضاء البيانات/عناصر المصفوفة a بالمدخلات.
- init : يقدم قيمة إبتدائية لعضو البيانات i ما تصطلح عليه التسمية بالإطلاق initialization.
- print : طباعة عناصر المصفوفة (عضو البيانات في الفئة stack)، حسب العناصر المدخلة.

إن من المستحسن دائما إعطاء قيمة إبتدائية للعناصر الأعضاء (data members) بمجرد تعريف الكائن، وذلك لأن القيم الابتدائية تساعد على استقرار المتغير المحجوز في الذاكرة وسهولة التعامل معه، هناك دالة عضوة في الفئة تقدمها (C++)، تقوم بعمل إطلاق (intialization) قيم إبتدائية لأعضاء بيانات في الفئة، وذلك بإستدعائها بعد التصريح عن الكائن في الدالة الرئيسية، هكذا:

```
stack s;
s.init();
```

إن الدالة ;()init تقوم بدور الدالة الباني كما سيأتي، مع فارق بسيط هو أننا ـ في حالة الباني ـ لن نحتاج بعد ذلك إلى إستدعاء دالة إضافية في الفئة .. إذا يتم تنفيذ الدالة العضوة (دالة الباني) بمجرد تعريف الكائن .. فالسطرين السابقين في البرنامج اعلاه، سوف يتم إختصارهما بسطر واحد هو:

```
stack s;
```

وكي نستفيد من هذه الفكرة في تعديل البرنامج السابق .. نحتاج إلى تعريف دالة باني في جسم الفئة بنفس محتوى الدالة ()init.

التصريح عن الباني وتعريفه Declaring and defining constructors:

يتم تعريف الباني عن طريق دالة عضوة (member function) ضمن الفئة الاصلية بنفس إسمها كالمثال التالي :

```
Stack()
{
I=0;
}
```

في المثال السابق يتم إستبدال الدالة العضوة ()init بالدالة العضوة أعلاه، إن هذه الدالة ()stack لا تحتاج إلى إستدعاء لأنها تنفذ بشكل طبيعي <u>بمجرد تعريف كائن وحجز موقع له في الذاكرة</u>، وبطبيعة الحال عند تعريف كائن آخر من نفس الفئة، يتم تنفيذ الدالة العضوة الخاصة به وإطلاق محتواها.

الدالة السابقة في المثال، تنفذ بمجرد تنفيذ السطر :

```
stack s;
```

تكمن أهمية الباني constructor في عملية الإطلاق التلقائي للقيم، الذي يسهل عملية التعامل مع الذاكرة، كما هو الحال مع أهمية إطلاق قيم إبتدائية للمتغيرات القياسية فالجملة التالية:

```
Int x=3; //the same statement like int x (3);
```

أفضل بكثير ـ كم ذكرنا سابقا ـ من المقطع البرمجي التالي:

```
Int x;
x=0;
```

ومن ناحية أخرى فإن مبدأ التغليف الذي يساعد على حماية محتويات الفئة يشجع على إستخدام الباني (وكذلك الهادم كما سيأتي) من أجل مأمونية التعامل مع البيانات الاعضاء الخاصة بالفئة .. ونذكر أن تطبيق مبدأ التغليف encapsulation يعني :

١. حماية أكثر للعناصر الأعضاء.
٢. أخطاء أقل من قبل المبرمج نفسه، أو الفئات الآخرى.

ولأن الباني هو دالة تنفذ بمجرد التصريح عن كائن، فإننا نكون مقيدين بشروط دالة الباني عند التصريح عن كائن من تلك الفئة، وذلك من جهة كون الدالة تحتاج إلى مدخلات (بارميترات) أم لا .. وبعدد المتغيرات المطلوبة، كما سيأتي.

نلاحظ أننا نستطيع تعريف الباني خارج إطار تعريف الفئة بالطريقة التي تعلمناها سابقا وذلك بإستخدام المؤثر (::) كالتالي:

```
stack::stack ()
{
I=0;
}
```

خصائص الباني characteristics of constructer :

a) يعتبر الباني دالة خاصة عضوة في الفئة، تسمح لنا بإطلاق قيم إبتدائية عند التصريح عن الكائن.
b) يتم إستدعاء دالة الباني تلقائيا (آليا) بمجرد التصريح عن الكائن.
c) بشكل إجباري يتم تسمية الدالة الباني بنفس تسمية الفئة، واي دالة (عضوة في فئة) تحمل إسم الفئة فهي دالة باني.
d) لا تحمل دالة الباني أي مخرجات، ولا حتى من النوع void، كإصطلاح تعريف.
e) الدالة الباني دائما تأخذ المدى public، ... (لماذا؟).
f) رغم كون دالة الباني لا تحمل اي مخرجات .. إلا أنها تأخذ اي نوع من المدخلات .. سواء مدخلات صفرية (zero-arguments)[1] أو مدخلات متعددة (مدخل واحد أو أكثر) -parameterized arguments .
g) يحق لنا وضع قيم تلقائيا (default value) لمدخلات دالة الباني، كما تعلمنا بخصوص أي دالة أخرى.
h) كذلك وكأي دالة أخرى .. يمكن لنا جعل دالة الباني متعددة الأسماء (overloading).

ترتيب تنفيذ دالة الباني order of constructor invocation:

في وقت التصريح عن كائن يتم في البداية حجز موقع في الذاكرة للكائن نفسه، ومن ثم يبدأ إستدعاء دالة الباني، ويتم التصريح عن دوال الباني للكائنات العامة (gloabal scope)أولا، وذلك حسب ترتيب التصريح عنها، البرنامج التالي يبين بعض هذه التفاصيل:

```
#include <iostream.h>
class sample
{
private:
        Int i;
public:
sample(int ii)
{
I=ii;
Cout<<endl<<"constructed"<<I;
}
```

[1] ـ المصطلح zero-arguments يعني عدم وجود مدخلات للدالة.

```
};
sample s1 (1);
sample s2(2);
void main()
{
sample s3 (3);
sample s4(4);

};
```

أما عملية الهدم destructed (تنفيذ دالة الهادم) فتتم بطريقة معاكسة لعملية البناء، اي يتم البدء من الكائنات المحلية ثم العامة (سنأتي لمفهوم الهادم لاحقا).

أنواع دوال الباني types of constructors :

١) الباني التلقائي (default constructor) : عندما لا نقوم بتعريف دالة باني لفئة ما، مالذي يحصل عند التصريح عن كائن من نوع تلك الفئة؟، كما حصل معنا في البرامج السابقة.
إن المترجم يقوم بنفسه من تعريف دالة باني (بدون أي محتوى)، تقوم هذه الدالة بالعمل فور التصريح عن أي كائن .. إن هذا النوع من دوال الباني يسمى بدالة الباني التلقائي، وهي دالة بدون مدخلات ولا محتوى .. فقط من الشكل:

```
Function ()
{}
```

ولكن عند تعريف أي دالة باني بواسطة المبرمج فإن المترجم لا ينشى دالة الباني التلقائية، سواء كانت دالة الباني الخاصة بالمستخدم تحتوي على مدخلات صفرية (zero-argumetns) أو متعددة المدخلات (parameterized-arguments)، أو حتى عند إنشاء اكثر من دالة باني بطريقة تعدد الاسماء (overloading)[٢] كما سيأتي .

٢) الباني متعدد المدخلات parameterized :
أحيانا نكون في حاجة إلى إطلاق قم إبتدائية مختلفة لكل كائن على حده (من نفس نوع الفئة)، فلو كان لدينا على سبيل المثال الفئة طالب، وكان الطالب الأول يبدأ بقيمة إبتدائية تشكل درجته في مادة، وكان الكائن الثاني (الطالب الثاني) يبدأ تصريحه بقيمتين إبتدائيتن هم درجته في مادة مثلا و قيمة مصروفه اليومي .. وكان الطالب الثالث لا يحتاج إلى اي قيم إبتدائية.
إننا في هذه الحالة نحتاج إلى اكثر من دالة باني .. تحتمل كل واحدة عدد مختلف من المدخلات حسب الكائن الذي يستدعيها كما سيوضحه لنا المثال التالي:

```
#include "iostream.h"
class sample
{
private :
int i;
float f;
public:
sample()
{
i=0;
f=0.0;
}
```

[٢]- يتم ترجمة المصطلح overloading حرفيا إلى التحميل الزائد، ولكن المفهوم لا يحتمل هذه الترجمة الحرفية، إذا أن الترجمة المناسبة للمفهوم هو إعادة التسمية (أو تعدد الأسماء) كنوع من أنوع تعدد الاشكال ploymorphisim، كما سنأتي إليه لاحقا.

```
sample(int ii,float ff)
{
i=ii;
f=ff;
}
void print ()
{
cout<<endl<<i<<endl<<f;
}
};
void main()
{
sample s1;
sample s2 (10, 16.78)
}
```

لاحظ عملية تعريف الكائنين s1,s2 بطريقتين مختلفتين .. حسب دالة الباني التي يتم إستدعائها، يسمى الباني هنا بالباني متعدد الاسماء (overloaded constructor)، حيث يحمل اكثر من تعريف بنفس الاسم والفارق هنا (كما هو معروف في الدوال من هذا النوع) هو عدد المتغيرات للتمييز بين الباني والآخر عند الإستدعاء.

فالكائن الاول s1 قام بإستدعاء الباني الأول ()sample والكائن الثاني s2 قام بإستدعاء الباني الثاني sample(int,float).

الفائدة الأخرى من إستخدام الباني متعدد الاسماء (overloaded constructor)، هو تحاشي إطلاق بعض القيم لا نحتاج إلى إطلاقها في الوقت الحالي .. عند التصريح عن الكائن.

من المهم ملاحظة التالي:

١. عند تعريف ـ فقط ـ دالة باني من النوع متعدد المدخلات، فلا يحق لنا عندئذ التصريح عن الكائن بالطريقة التقليدية (sample s1;) إذ ان تمرير المتغيرات أصبح (في هذه الحالة) غير إختياري.

في هذه الحالة ولتجنب الوقوع في الخطأ نقوم بتعريف هادمين بطريقة تعدد الاسماء overloaded constructor، يكون أحدهما بدون مدخلات zero-arguments، وذلك لتحاشي هذه المشكلة المتوقعة .. كما فعلنا في المثال أعلاه.

٢. إن العبارة (sample s1 (2);) تكافئ تماما العبارة (sample s1=2;) ، وأي منهما تستخدم لعملية التصريح عن كائن يطلق دالة باني بمتغير/مدخل واحد .. كحالة محددة.

٣) باني النسخ copy constructor :

لنفترض أننا أردنا التصريح عن كائن وبعد مجموعة من العمليات عليه قمنا بنسخ محتوياته (آخر قيم للبيانات الأعضاء) إلى كائن جديد من نفس النوع.

سوف نستخدم الجملة:

```
callstype obj1;
........
.......
classtype obj2=obj1;
```

وذلك على إفتراض أن الكائن القديم هو obj1 وتم نسخ محتوياته إلى الكائن الجديد obj2.

ولكننا نعلم أن الجملة الأخيرة (classtype obj2=obj1;) هي عبارة عن إطلاق باني مفترض للفئة calsstype تقوم دالة هذا الباني بإستقبال مدخل واحد من نوع الفئة نفسه، ونسخ محتوياته إلى الكائن obj2 الذي اطلق هذه الدالة.

ولكن عند تطبيق مثال على البرنامج أعلاه نجد اننا لسنا بحاجة إلى كتابة دالة باني تقوم بهذه العملية، إنها دالة باني هامة يسمى باني النسخ(copy constructor)، يقوم مترجم لغة C++ بتنفيذها تلقائيا، دون أن يضطر المبرمج إلى كتابتها.

الفائدة الأخرى من دالة باني النسخ هي عملية تمرير كائن إلى دالة (أخرى .. سواء كانت عضوة في فئة أو لا)، خاصة إذا كان التمرير بطريقة التمرير بالقيمة (passing by value)، إن التمرير بالقيمة كما نعلم يقوم بأخذ نسخة من المتغير الممر ، وإجراء العمليات عليه .. في حالة كان المتغير الممر هو كائن ..فإن دالة باني النسخ تقوم بعملها تلقائيا، دون أن يصدع المبرمج دماغه بكتابتها.
و كذلك عملية إرجاع القيمة من دالة .. (Return Value) .. تمر بنفس عمليات النسخ سابقة الذكر.
السؤال الآن :
تمرين: قم بكتابة فئة لنوع طالب، تحتوي على عناصر أعضاء (كالعمر والدرجة مثلا)، وتحتوي على دالة باني واحدة تقوم بدور دالة باني النسخ وذلك بنسخ محتويات الكائن الحالي إلى كائن جديد.
مساعدة:
الكائن الممرر لدالة باني النسخ يجب ان يكون بطريقة التمرير بالمرجع (passing by reference) .. لماذا؟
وذلك لأن تمرير الكائن بطريقة التمرير بالقيمة .. يعني .. عمل نسخة من ذلك الكائن .. النسخة نفسها سوف تستدعي دالة الباني التلقائي .. مما يعني عدم إستخدام قيم الكائن الممر التي نريد نسخها أصلا.
٤) الباني الديناميكي (dynamic constructors):
هي دوال باني عادية .. لكنها تستخدم عناصر أعضاء من النوع المؤشر (pointer)، في هذه الحالة تقوم بعملية التعريف الديناميكي للمؤشر عند التصريح عن الكائن .. ويتم ذلك بإستخدام المؤثر الخاص بلغة C++ والمستخدم لحجز المتغيرات الديناميكية في الذاكرة.
نعلم جميعنا ان المتغيرات العادية (الاستاتيكية) يتم حجز موقعها في الذاكرة طوال عمل البرنامج، ولا يتم غالبا التخلص من هذا الحيز إلا عند إنهاء البرنامج.
إن المؤثر (new) - الذي لم يكن موجودا في لغة C، يقوم بعملية حجز موقع ديناميكي للمتغير .. يكون قابلا للإلغاء عن طريق المؤثر المقابل delete بالشكل:

x = new int;//التصريح عن المتغير الصحيح وحجز موقع ديناميكي له في الذاكرة
delete x;// عملية إلغاء موقع المتغير من الذاكرة

في المثال التالي نعرف مؤشر يؤشر إلى بداية مصفوفة بطريقة عادية مرة ، وبطريقة ديناميكية مرة أخرى .. ومع ذلك فكلا دالتي الباني هما من النوع الديناميكي لإستخدام الطريقة الديناميكية في حجز قيم البيانات الأعضاء في الذاكرة.

```
#include "iostream"
using namespace std;

class array
{
private:
int *a;
int dim,i;
public:
array() //zero-argument constructor
{
a=new int[10];
dim=10;//the dimension of the array
i=0;//determine the first element of the array
}
array(int l)
{
a=new int[l];
dim=l; //the dimension of the array
i=0;
}
```

```
void add(int d) //function member to fill the array
{
if (i>=dim)
{
cout<<endl<<"array boud exceed";
i=0;
return;
}
a[i]=d;
i++;
}
void print()
{
    for (int j=0;j<dim;j++)
    cout<<endl<<a[j];
}
};

void main()
{
    array a1,a2(5);

    a1.add(100);
    a1.add(300);
    a1.print();

    a2.add(1);
    a2.add(2);
    a2.add(3);
    a2.add(4);
    a2.add(5);
    a2.print();
}
```

التصريح عن الهادم وتعريفه Declaring and defining destructors:

إن دالة الباني تستدعي بشكل تلقائي عند التصريح عن الكائن، وبطريقة معاكسة فإن دالة الهدم تستدعي بمجرد إنتهاء الكائن أو إنقراضه!... كما سنوضح حالا..

إن الهادم هو دالة عضوة في الفئة تحمل نفس التسمية الخاصة بالباني مع إضافة مؤثر الهادم (~) قبل إسمها، فإذا كان إسم دالة الباني للفئة student هو كما نعلم student()، فإن دالة الهادم هي ~student() .

المثال التالي يوضح ذلك:

```
#include "iostream"
using namespace std;

class example
{
public:
        example()//constractor...
```

```
        {
                cout<<endl<<"*******inside the constructor******"<<endl;
        }
        ~example()//destructor..
        {
                cout<<endl<<"~~~~~~~inside the destructor~~~~~~~~"<<endl;
        }
};
void main()
{
        example e;
        cout<<endl<<"at first we get the constructor message"<<endl;
   cout<<endl<<"then before the program is ended we get the other message"<<endl;
}
```

إن تنفيذ الشفرة الموجودة في الدالة (~example()) يعني أن الكائن قد إنتهى من الذاكرة.
وكما هو الحال بالنسبة للباني، فإن الهادم لا يعطي أي مخرجات ولا حتى من النوع (void).
وكذلك نستطيع تعريف الهادم خارج إطار حاصرتي الفئة كالتالي:

```
example::~example()
{
}
```

وذلك كأي دالة عضوة عادية بعد التصريح عنها داخل الفئة بالصورة:

```
~example();
```

خصائص دالة الهادم characteristics of destructors :

a) يتم إستدعاء الهادم بشكل تلقائي عندما يذهب الكائن خارج المدى scope المحدد له، فإذا كان مدى الكائن محليا فإن دالة الهدم تنفذ عند حصول إرجاع على مستوى الكود المحلي، وإذا كان مدى الكائن عاما فإن دالة الهدم تنفذ بمجرد إنها البرنامج.
b) لا يقبل الهادم أية مدخلات (خلافا للباني)، كما أنه لا يقبل اي مخرجات كالباني.
c) لأن الهادم لا يقبل مدخلات فهو بالتالي لا يمكن ان نطبق عليه عملية تعددت ألاسماء overloading .. وعليه فإنه يمكن ان يكون للكائن اكثر من باني .. ولكن لن يكون للكائن غير هادم وحيد.
d) يستخدم الهادم لمعرفة خروج الكائن عن السيطرة .. ولكننا لا نستطيع إستدعاؤه للقيام بعملية (هدم) الكائن.

+++++++++++++++++++++

-٣-

التحميل الزائد للمؤثرات Operator Overloading

ماهو التحميل الزائد للمؤثرات؟
يعتبر التحميل الزائد او مبدأ إعادة التسمية واحد من المميزات الرائعة في لغة C++، كلغة برمجة تعتمد على نظرية التوجه الكائني/الهدفي/الشيئي!.

المؤثر operator هي رمز معتمد في لغة البرمجة يختلف عن الكلمات المفتاحية key word في كونه لا يحتوي على حروف المعجم، ومن امثلة المؤثرات : مؤثر الجمع + مؤثر الضرب * مؤثر باقي القسمة % .. وباقي الرموز الرياضية والمنطقية التي تجمعها التسمية مؤثرات.

بواسطة مبدأ التحميل الزائد، نستطيع إعطاء هذه المؤثرات معني إضافية، تختلف او تضيف ميزات أخرى للتعريفات الأصلية التي تعودنا عليها في البرمجة بلغة C++.

هذه المؤثرات عودتنا على صيغة معينة للتعامل معها، وذلك بالتعود على أنواع البيانات التقليدية التي نعرفها، فمؤثر الجمع مثلا، يتعامل مع الأعداد الصحيحة والعائمة (الحقيقية) بطريقة معروفة .. ولكن ماذا لو أردنا إستخدامه في المتغيرات النصية أو المصفوفات النصية، فلو كان لدينا المصفوفتين النصيتين str1,str2 وكانت قمية الأولى "ahmad" وقيمة الثانية "basem"، فلو اردنا الحصول على النتيجة str1+str2 لقمنا بكتابة الكود التالي:

```
char str1[20]="ahmad";
char str2[]="basem";
char str[20];
strcpy(str3,str1);
strcat(str3,str2);
```

المقطع البرمجي السابق يقوم بنسخ str1 إلى str3، ثم يقوم بإضافة محتوى str2 إلى str3، لتكون قيمة str3 آخر المطاف هي ("ahmadbasem")، لا غبار على اننا نفذنا المطلوب ولكن اليس الشكل المألوف :

```
str3=str1+str2;
```

أفضل في التعامل على المدى البعيد.

إن مبدأ التحميل الزائد للمؤثرات يعني من الناحية العملية ، عملية إعادة تعريف المؤثرات المعروفة لتتعامل مع الأنواع الجديدة والغير قياسية، كمصفوفة النصوص في المثال أعلاه .. ونوع البيانات الخاص بالمستخدم هو كما نذكر دائما هو الفئة التي يقوم بإنشاءها .. فيستطيع عندئذ تعريف المؤثرات التي يريد ان يراها ضمن الفئة الخاصة به.

إن التحميل الزائد للمؤثرات هو أحد تطبيقات مبدأ تعدد الأشكال (polymorphism)، وهو يعني بإختصار إضافة عملية جديدة للمؤثر بالإضافة إلى العمليات السابقة التي يقوم بها.

التحميل الزائد للمؤثرات الأحادية overloading unary operations:

المؤثرات الأحادية هي مؤثرات تأخذ متغيرا واحد، فعملية الجمع مثلا مؤثر ثنائي في الأساس.. ولكنها تأخذ في لغة C++ شكل المؤثر الأحادي في الجملة:

```
X++;
```

وكذلك الحال مع المؤثر الأحادي(--) .

المثال التالي يعرف المؤثر الأحادي ++، على عناصر وكائنات الفئة index :

```
#include "iostream"
using namespace std;
class index
{
private:
  int count;
public:
  index()
  {
        count=0;
  }
  void operator++()
  {
        ++count;
  }
  void showdata()
  {
```

```
            cout<<count;
      }
    };

    void main()
    {
      index c;

      cout<<endl<<"c=";
      c.showdata();

      ++c;
      cout<<endl<<"c=";
      c.showdata();

    ++c;
      cout<<endl<<"c=";
      c.showdata();
    }
```

إن التحميل الزائد هنا يعلم المؤثر ان يتعامل مع أنواع البيانات المعرفة بواسطة المستخدم، في المثال السابق صار المؤثر ++ يتعامل مع نوع البيانات الجديد index، وذلك بتعريف دالة عضوة في تلك الفئة بإستخدام الكلمة المحجوزة operator، ويليها إسم المؤثر المراد التعامل معه.

بخصوص الجملة :

```
void operator++()
```

لاحظ الكلمة المحجوزة operator، المتبوعة بالمؤثر ++ هي الطريقة البرمجية لتطبيق مبدأ التحميل الزائد للمؤثرات.

في المثال السابق تم إطلاق قيمة العضو count بالقيمة صفر عن طريق الدالة الباني ()index، وخلال كود البرنامج تم زيادة العنصر بتطبيق المؤثر ++ على الكائن c نفسه .. المصرح عنه في البرنامج.

المخرجات المتوقعة للكود أعلاه هي:

```
c=0;
c=1;
c=2;
```

بالنسبة لطريقة تنفيذ الجملة c++، فإنها تنفذ ضمن الفئة كأي دالة عضوة أخرى بالشكل:

```
c.operator++();
```

عند تنفيذ تلك الدالة .. لا توجد مدخلات كما أنه لا توجد مخرجات ـ كما هو واضح، يفرق المترجم بين العبارتين c++ والعبارة count++، حسب نوع المتغير المسند للمؤثر .. فإذا كان كائنا من الفئة index نفذ الدالة ()c.operator++ وإذا كان النوع قياسي (int أو float أو double مثلا) فإنه يتعامل معه حسب التعريف المسبق المدمج باللغة.

والآن مالذي سوف يحدث لو كتبنا الجملة:

```
d=++c;
```

حسب تعريف الدالة العضوة في الفئة index، نحن جعلنا مخرجاتها void، هذا يعني ان تنفيذ الجملة السابقة مستحيل.. سوف يعترض المترجم compiler، وربما أعطى رسالة كالتالي:

error C2679: binary '=' : no operator found which takes a right-hand operand of type 'void'

يتضح من رسالة الخطأ أعلاه أن المترجم يطالب بأن يكون للدالة العضوة نوع إرجاع من نوع المتغير d، أي كائن من النوع indexs.

نحتاج إذن إلى تعريف الدالة العضوة (()++void operator) بطريقة أخرى، تكون لدينا عندئذ مخرجات، ولنعدل الآن البرنامج السابق بحث يبدو كالتالي:

```
#include "iostream"
using namespace std;
class index
{
private:
    int count;
public:
    index()
    {
            count=0;
    }

    index operator++()
    {
            ++count;
            index temp;
            temp.count=count;
            return temp;
    }
    void showdata()
    {
            cout<<count;
    }
};

void main()
{
    index c,d;

    cout<<endl<<"c=";
    c.showdata();

    ++c;
    cout<<endl<<"c=";
    c.showdata();

    d=++c;
    cout<<endl<<"c=";
    c.showdata();
    cout<<endl<<"d=";
    d.showdata();
}
```

في هذا التعديل على البرنامج، قمنا في دالة التحميل الزائد بإجراء عملية الزيادة على المتغير الخاص بالكائن الأصلي (c في هذه الحالة)، ومن ثم قام بنسخها إلى كائن وسيط هو temp، ليقوم بعد ذلك بنسخ قيمة المتغير الجديدة من temp إلى الكائن d، وذلك من أجل تطبيق الجملة ;d=++c.
سيكون مخرجات البرنامج أعلاه كالتالي:

```
c=0;
c=1;
c=2;
d=2;
```

وهناك طريقة أخرى لإنجاز دالة التحميل الزائد (index operator++())، وذلك بكتابتها بالطريقة التالية:

```
index operator++()
{
++count;
Return index (count);
}
```

ولكن هذه الدالة لن تنفذ بصورة صحيحة مالم نعرف .. دالة باني جديدة تقبل المتغير count هنا، لتكون بالشكل:

```
index (int i)
{
count=I;
}
```

وعلى أساس هذين التغيرين سوف يبدو البرنامج أعلاه بالصورة:

```
#include "iostream"
using namespace std;
class index
{
private:
   int count;
public:
   index()
   {
          count=0;
   }
   index (int i)
{
count=i;
}
   index operator++()
{
++count;
return index (count);
}

   void showdata()
   {
          cout<<count;
   }
};
```

```
void main()
{
   index c,d;

   cout<<endl<<"c=";
   c.showdata();

   ++c;
   cout<<endl<<"c=";
   c.showdata();

   d=++c;
   cout<<endl<<"c=";
   c.showdata();
   cout<<endl<<"d=";
   d.showdata();

}
```

نستطيع كذلك إستبدال السطرين :

```
++count;
return index (count);
```

بسطر واحد هو :

```
return index (++count);
```

أخر ما يمكن اضافته في موضوع المؤثر الأحادي هو الكود اللازم من أجل المؤثر العكسي، ;c++ وهذا يعني ان نكتب دالة تحميل زائد من أجل المؤثر المعاكس للمؤثر c++ وهو المؤثر (c++) ولكي نفعل ذلك لنضف الدالة التالية فقط للبرنامج السابق:

```
index operator++(int)
{
return index (count++);
}
```

للتذكير فإن الفارق بين الجملتين (c++) و(c++)، هو ان التزايد في الجملة الأولى يحدث قبل اسناد القيمة الأصلية، بينما في الثانية يحصل بعد إسناد القيمة الاصلية للمتغير .

سؤال:(مالفارق بين الجملتين (d=++c) و(d=c++))؟

سؤال آخر : ما الذي سوف يحدث لو لم نقوم بإضافة الدالة الأخيرة لتعريف الفئة ، ثم إستخدمنا الاسناد: d=c++ ؟

لإجابة السؤال الثاني علينا إضافة الجملة في السؤال إلى البرنامج أعلاه .. ثم نكتب دالة المؤثر المعاكس ولنقارن بين المخرجات للبرنامجين.

ملاحظة :

١.إن عدم كتابة دالة التحميل الزائد للمؤثر العكسي لن تجعل المترجم يرفض التعبير بالمؤثر المعاكس ولكنه سوف يتعامل معه كأنه نفس المؤثر المعرف أعلاه.

٢. إن الصيغة index operator++(int) لا تعني بالضرورة أن هذه الدالة تحتمل مدخلات من النوع الصحيح .. وذلك لأن دالة التحميل الزائد للمؤثر الأحادي لا تقبل مدخلات كما لاحظنا مسبقا..ولكن العبارة السابقة تستخدم الكلمة المحجوزة int لتنبيه المترجم بأن التحميل الزائد هنا يختلف عن السابق .. بطريقة معاكسة من أجل تعريف المؤثر المعاكس.

التحميل الزائد للمؤثرات الثنائية overloading binary operators :

لتعريف عمليات جديدة على المؤثرات الثنائية .. مثلا مؤثر الجمع + .. نحتاج إلى إستخدام نفس الكلمة المحجوزة operator ضمن دالة التحميل الزائد .. ولكننا هنا سوف نستخدم كائن من نفس الفئة كمدخل للدالة .. ونستخدم كائن آخر لإرجاع النتائج.

كمثال على ذلك : البرنامج التالي يجري عمليات الجمع والضرب بين الأعداد المركبة (complex numbers)، وحين نعرف كائن كعدد مركب فهو يأخذ عضوي بيانات .. الأول يمثل الجزء الحقيقي للعدد .. والثاني يمثل الجزء التخيلي للعدد.

وللتذكير فإن العدد المركب هو العدد من الشكل :

c=x+iy ⇔ c=(x,y)

حيث أن العدد x هو عدد حقيقي يمثل الجزء الحقيقي من العدد، والعدد y هو عدد حقيقي ايضا ولكنه يمثل الجزء التخيلي للعدد المركب ..ولإجراء عملية الجمع بين عددين، فإننا نقوم بجمع الجزئين الحقيقيين على حده .. كما نجمع الجزئين التخيليين على حده.

أما عملية الضرب فتتم بشكل أعقد ولتوضيح العمليتين ليكن لدينا العددين المركبين c1 و c2 المعرفين كالتالي:

c1 = x1 + i y1 ;
c2 = x2 + i y2 ;

فإن:

c1 + c2 = (x1 + x2) + i (y1 + y2)
c1 * c2 = (x1 * x2 - y1 * y2) + i (x1 * y2 - x2 * y1)

المطلوب هنا تعريف دالة تحميل زائد للمؤثرين +، * من أجل إنجاز العمليات السابقة ضمن كائنات من نوع الفئة الخاصة بالأعداد المركبة، وهذا ما يوضحه البرنامج التالي:

```
#include "iostream"
using namespace std;
class complex
{
private:
   float real,image;
public:
   complex()
   {}
   complex (float r,float i)
   {
          real=r;
          image=i;
   }
   void setdata(float r,float i)
   {
          real=r;
          image=i;
   }
   complex operator+(complex c)
   {
          complex t;
          t.real=real+c.real;
          t.image=image+c.image;
          return t;
   }
   complex operator*(complex c)
   {
```

```
            complex t;
            t.real=real*c.real-image*c.image;
            t.image=real*c.image+c.real*image;
            return t;
        }
        void display(char *a)
        {
            cout<<a;
            cout<<endl<<"real="<<real<<endl;
            cout<<"imaginary="<<image<<endl;
        }
    };
    void main()
    {
        complex c1,c2(2.5,3.9),c3;
        c1.setdata(5,9.3);

        c1.display("c1");
        c2.display("c2");

        c3=c1+c2;
        c3.display("c3");

        c1=c2*c3;
        c1.display("c1");
        c2.display("c2");

        c3=c2+c1*c3;
        c3.display("c3");
    }
```

يتم إدخال بيانات الكائن من فئة الـ complex، بإحدى طريقتين إما عن طريق الباني، وذلك بإسناد القيمة مباشرة مع التصريح عن الكائن .. كما مع الكائن c2، أو عن طريق الدالة ()setdata التي يمرر لها العنصرين التخيلي والحقيقي للكائن.

تقوم الدالة ()display بعد ذلك بعرض القيم الحقيقية والتخيلية للكائن المطلوب .. لفحص التغيرات التي حصلت عليه.

الهدف الاساسي من هذا البرنامج هو معرفة كيف يتم كتابة كود دالة التحميل الزائد لمؤثر ثنائي .. والمثال المعروض هنا هما المؤثر + والمؤثر * بالنسبة للأعداد المركبة.

في هذا المثال وقبل ظهور مبدأ التحميل الزائد للمؤثرات كان المبرمج سيقوم بتعريف دالة عضوة خاصة بالجمع، ويسميها مثلا (add_complex) و يعرف دالة للضرب ويسميها مثلا(mul_complex)، وسيكون عندئذ تطبيق الجملة:

```
s3=s1+s2;
```

في هذه الحالة بالشكل:

```
c3=(c1. add_complex(c2))
```

إذن كيف سيكون شكل الجملة ;s3=s1+s2*s3 ؟!!

والآن لنعد إلى دالة التحميل الزائد للمؤثر + :

```
complex operator+(complex c)
{
```

```
                complex t;
                t.real=real+c.real;
                t.image=image+c.image;
                return t;
        }
```

عند إستدعاء المؤثر + (أي إستدعاء هذه الدالة)، فإن الكائن الثاني c2 يمرر لها ويجمع مع عناصر الأول، وتصبح الجملة :

```
s3=s1+s2;
```

وكأنها :

```
s3= s1.operator+(s2);
```

بحيث يعود المتغير real في الدالة العضوة إلى الكائن s1، أما المتغير c.real فهو يعود بالطبع إلى الكائن s2.

المؤشر (*this*) the ***this*** pointer :

إن عملية الربط بين الدوال الأعضاء لكائن والبيانات الأعضاء للكائن تتم عن طريق مؤشر تلقائي يتم تعريفه بمجرد التصريح عن الكائن .. هذا المؤشر يشير إلى موقع الكائن نفسه في الذاكرة .. وتصل إليه الدوال الأعضاء بطريقة مباشرة .. وذلك من أجل التعامل مع البيانات الأعضاء للكائن.

يستعمل من أجل هذا المؤشر الكلمة المحجوزة this التي تعني المؤشر الذي يشير إلى عنوان الكائن في الذاكرة .. ولمعرفة طريقة إستخدام المؤشر لننظر إلى البرنامج البسيط التالي:

```
#include "iostream"
using namespace std;
class example
{
private:
   int i;
public:
        void setdata(int ii)
        {
                i=ii;// a way to set the data;
        }
        void showdata()
        {
                cout<<i;
        }
};
void main()
{
        example e;
        e.setdata(10);
        e.showdata();
}
```

يقوم هذا البرنامج البسيط جدا .. بتعبئة بيانات اعضاء بطريقة مباشرة وواضحة .. ثم يطبعها.

نستطيع إستبدال بعض الأسطر في هذا البرنامج بطريقة مكافئة عن طريق إستخدام المؤشر *this* .. كالتالي:

```
#include "iostream"
using namespace std;
class example
{
```

```
private:
   int i;
public:
        void setdata(int ii)
        {
                cout<<"my objects address is "<<this<<endl;
                this->i=ii;
        }
        void showdata()
        {
                cout<<"my object address is "<<this<<endl;
                cout<<this->i<<endl;
        }
};
void main()
{
        example e;
        e.setdata(10);
        e.showdata();
}
```

الملاحظ هنا إستخدام التعبير this->i بدلا من i نفسه .. وهي طريقة مكافئة تشبه التعبير obj.i، مع فارق إستخدام المؤشر هنا .. كذلك يستخدم المؤشر this لمعرفة عنوان الكائن المصرح عنه .. e في هذه الحالة.
للمؤشر this إستخدامات هامة سنتعرف على بعضها لاحقا.

التحميل الزائد لمؤثر المقارنة overloading comparison operators:
الامثلة السابقة قد أوضحت بطريقة عملية طريقة التحميل الزائد لأهم المؤثرات .. و من أجل المقارنة بين كائنين من النوع نفسه .. نقوم بتعرف دالة < operator حسب طريقة المقارنة المطلوبة .. فمثلا بخصوص مثال الفئة complex سوف نعرف المقارنة حسب النموذج التالي :
ليكن لدينا العددين المركبين z1,z2 يكون العدد z1 أكبر من العدد المركب z2 إذا كان :

z1 = x1 + i y1 ; z2 = x2 + i y2;

وكان :

x1>x2 and y1>y2 ➔ z1>z2;

يتم تنفيذ التعريف السابق بالكود التالي :

```
#include "iostream"
using namespace std;
class complex
{
private:
        float real,image;
public:
        complex()
        {}
        complex (float r,float i)
        {
                real=r;
                image=i;
```

```
        }

        int operator>(complex c)
        {
                if (real>c.real && image>c.image )
                        return 1;
                else
        return 0;
        }
};
void main()
{
        complex c1(3,4),c2(2.5,3.9);
        if (c1>c2)
                cout<<"OK"<<endl;
        else
                cout<<"conflict";

}
```

يقدم هذا الكود صيغة المقارنة أكبر من < ، ولكن ماذا لو تم طلب المقارنة > اصغر من .. بالتأكيد سوف يعترض المترجم لأنه لن يفهم التحميل الزائد للمؤثر ما لم يتم كتابة الكود اللازم ..

تمرين : قم بكتابة الكود اللازم لمؤثرات المقارنة الثلاثة المتبقية (=<,=>,<) ؟

+++++++++++++++++++++

-٤-

الوراثة والكائنات المشتقة Inheritance and Derived Classes

الوراثة في المبدأ الكائني Inheritances in OO Methodology:

تعتبر الوراثة Inheritance حجر الزاوية في المبدأ الكائني بصورة عامة، سواء في البرمجة الكائنية OOP، أو في التصميم الكائني OOA، أو في التحليل الكائني OOD، ويتم إستخدام مبدأ الوراثة من أجل تحقيق مجموعة كبيرة من الفوائد للمبرمج ومصممي البرامج بصورة عامة من جهة، ولمستخدمي البرامج المخلتفة من جهة أخرى.

ويمكن تعريف الوراثة في البرمجة الكائنية بأنها عملية التصريح عن كائن إبن (child)/كائن مشتق (derived)، يحمل هذا الكائن الجديد مجموعة من الخصائص (البيانات الاعضاء data members) والخدمات (الدوال الاعضاء member functions)، وبالاضافة إلى ذلك (يرث) هذا الكائن الابن او المشتق جميع صفات (خدمات وخصائص) الكائن الاب (parent)، واحيانا يسمى بالكائن الاساس Base Calss كما يوضحه المثال التالي:

لدينا الكائنين (أحمد) و(فاطمة) يحملان صفات كثيرة مشتركة، ولكنهما يختلفان في خصائص أخرى، سوف نستخدم لغة النمذجة الموحدة UML، في تمثيل الكائنين كفئتين منفصلتين .. ونعرف كذلك الفئة إنسان كما بالشكل (١)، أما الشكل التالي (الشكل ٢) فيرمز إلى عملية توريث صفات الفئة إنسان إلى كلا الفئتين ذكر وأنثى، فنقوم عندئذ بتعريف الكائنين علي وفاطمه بشكل منفصل، ولكنهم سوف يشتركان بالصفات الموجودة في الفئة إنسان.

فاطمة
العمر ٤٥ الجنس "انثى" الطول ١٥٠
يأكل () يشرب() يلد () يرضع ()

أحمد
العمر ٣٢ الجنس "ذكر" الطول ١٦٧
يأكل () يشرب () يتزوج () يحارب ()

انسان
العمر الجنس الطول
يأكل () يشرب ()

شكل (١)

واضح جدا ان المثال في الشكل (١) عبارة عن مثال توضيحي فقط لفكرة الوراثة، نفترض ان فاطمة وأحمد يشتركان في الخصائص الموضحة في الشكل، وأنهما يختلفان في الخدمات والاعمال المذكورة في الشكل، لو اردنا ان نعرف فئة اساسية للكائنين لكانت الفئة إنسان، ولكن الفئة انسان تختلف في كونها لا تستطيع ان تحمل الخدمات المختلفة بين الكائنات المختلفة.

إن حل المشكلة موضح في الشكل (٢)، حيث صرحنا عن ثلاث فئات: الاولى الفئة (إنسان) كفئة أب/ اساس، والفئتين الابناء او المشتقتين ذكر وأنثى.

انسان
العمر الطول
يأكل () يشرب ()

أنثى
يأكل () يشرب() يلد () يرضع ()

ذكر
يأكل () يشرب() يحارب ()

شكل (٢)

من الشكل (٢)، نلاحظ ان الفئتين انثى وذكر ترثان جميع صفات الفئة انسان ، بالإضافة إلى ان الفئة الابن سواء ذكر او انثى تحمل صفات خاصة بها، بالإضافة إلى كونها تملك جميع صفات الفئة العليا.
إن الفئة الإبن تعيد إستخدام (reuse) الصفات الموجودة في الفئة العليا، وهي بالإضافة إلى ذلك تملك الصفات الفريدة الخاصة بها.

تعريف فئة مشتقة بلغة ++C :

يتم تعريف الفئة المشتقة بلغة السي ++، عن طريق الشكل العام التالي:

```
class derived_class : visibility-mode base_calss
{
        Derived calss methods and members;
};
```

class : كلمة محجوزة لتعريف الفئات بشكل عام.
derived_class إسم المتغير الذي سوف تأخذه الفئة المشتقة.
: مؤثر تعريف الفئة المشتقة.
visibility-mode : عبارة إختيارية تحدد مدى الرؤية وهي إما (private) أو (public) أو (protected)، وهي تعرف مدى الرؤية بالنسبة للبيانات الاعضاء الموروثة من الفئة الاساس/الاب.
base_calss : اسم الفئة الاساس او الفئة الاب، ويجب بالطبع ان تعرف بشكل مسبق قبل تعريف الفئة المشتقة.
....
....
....

+++++++++++++++++++++++